APPROXIMATION OF CPU CODE USING NEURAL NETWORKS

BY

CONOR GARDNER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Nam-Sung Kim

ABSTRACT

There is a well-known spectrum of computing hardware ranging from central processing units (CPUs) to highly specialized application specific integrated circuits (ASICs). Most consumer CPUs are general purpose and come with mature development tools used by large communities of programmers, while ASICs can perform very specific tasks very efficiently at the expense of ease-of-use and flexibility. Other devices such as digital signal processors (DSPs), graphics processing units (GPUs), and field programmable gate arrays (FPGAs) occupy intermediate interpolations on the usability-efficiency continuum.

New development tools such as very long instruction word (VLIW) compilers, CUDA, and logic synthesis have made it easier than ever for even novice programmers to leverage the increased efficiency of DSP cores, GPUs, and FPGAs using specialized high-level programming languages for those devices. However, even after surmounting the steep learning curve, a skilled programmer will still require significantly more time to write and validate a CUDA or OpenCL function compared to an equivalent CPU function.

Neural nets are fairly general purpose tools which can perform pattern recognition or arithmetic operations on a block of input data and produce a corresponding block of output data. The aim of this project is to be able to select a fairly arbitrary block of code such as a C++ function and train a neural net to mimic the original code's input-output behavior. Once the neural net has been trained, it can run on a highly parallel device such as a GPU without the programmer ever needing to write a CUDA program. Of course, this approach also has inherent drawbacks. First, all dependent processing which consumes output data from the neural net must be able to tolerate errors, since the network can only approximate the original code. Second, since neural nets require many, often unnecessary, floating point operations, there will be a large amount of “bloat” in the neural implementation which must be offset by the benefits gained by running the workload on a highly parallel device to be practical.

ACKNOWLEDGMENTS

This project was a cooperative effort with major contributions from Yu-Hsuan Tseng and Nam-Sung Kim.

TABLE OF CONTENTS

CHAPTER 1: NEURAL NETWORK BACKGROUND

1.1 Introduction

Artificial neural networks are not as new as many engineers believe, as they were first used as early as 1954 by Farly and Clark who implemented a Hebbian network. In recent years, there has been a large renewal of interest in neural nets partially due to advancements in the backpropagation algorithms used for training and an increased availability of computing resources which have made neural nets practical for anyone with a consumer grade computer. Unfortunately, neural nets also are considered somewhat “magical” by most programmers, even those programmers who are successful at applying them. Therefore, although the content of this chapter is not particularly novel, it serves the important role of demystifying what neural nets are and how they work internally with the hope that this background knowledge will make it easier for the reader to understand the more advanced topics addressed later. The material presented by the Tensorflow tutorial [4], is similar to that of chapter 1 and is an excellent source for further reading. In addition to presenting background information, chapter 1 aims to provide insight into what neural networks can and cannot do, their computational costs, and how to choose good training data.

1.2 Motivation for Neural Networks and Basic Architecture

Neural networks are not suited for every task, but they do excel at certain image processing operations such as recognizing specific features or sub-images within a large image, which was previously impossible to perform reliably. At first glance, this problem seems easy but it is actually quite frustrating due to the unavoidable and extreme variations in lighting, viewing angle, occlusion by other objects, and size between perfectly identical items within different images. Additionally, realistic objects such as eyes, coins, and even bar codes will have true variation such as color differences, differences in shape, scratches, dirt, and abnormalities. Being able to reliably classify an image as a “face” is quite challenging since there are many

visually different types of valid faces as well as a devious arsenal of images which may look similar to faces and produce false positives.

Suppose you are building an artificial intelligence (AI) vision system that continuously processes frames generated by a camera in real time. Now suppose that you wanted to recognize a specific class of object, such as a face, whenever it is present in the video stream. A basic face detection algorithm could take a single frame from the camera as input and produce an output image of the same dimensions such that each pixel contains a single value between 0 and 255 indicating the certainty of a face existing at that location. Each output pixel flags whether a face exists at that location or not. To calculate the value of each pixel in the output, a square tile from the input image centered on the X-Y location of the output pixel is examined. The square tile could then be compared to an ideal or average image of the face you want to detect. The closer the match between the pixels from the input tile and those of the ideal face image, the closer to 0 that output pixel will be.

Consider the following simple algorithm which computes the above metric. The image we want to detect within the larger input image is typically referred to as the *kernel*. First, for each output pixel, an input tile from the input image is defined such that the input window is centered on the X-Y location of the output pixel and is the same size as the kernel. Next, each pixel from the input tile is subtracted from the pixel at the same location in the kernel image. Finally, the absolute value of each subtraction is taken, and these absolute values are averaged to yield the output pixel. Again, each output pixel computes a metric of how closely the neighborhood around that pixel from the input image resembles the kernel image.

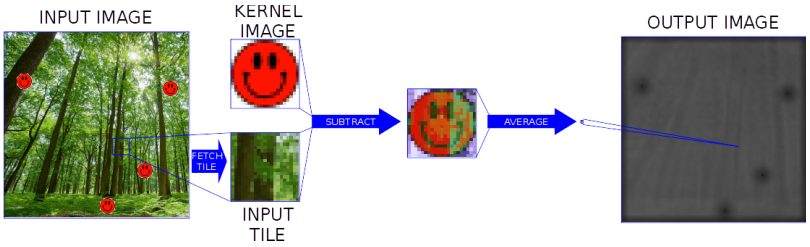


Figure 1. Operations similar to convolution can help detect specific features in images. In this case, darker output pixels indicate stronger matches.

The image processing algorithm described here and illustrated in figure 1 is similar to image convolution. However, convolution as defined in standard mathematics would multiply each pixel from the input tile and kernel image, rather than take the absolute difference. Additionally, convolution would technically call for the kernel image to be flipped, but this would only be an inconvenience to the programmer. In practice, true convolution and the operation used here produce similar looking images if the kernel is properly normalized. Notice the dark black patches in the output image in the regions where the face was present in the input image.

Additional convolutional layers could be running in parallel with each other to detect other body parts such as arms or a torso. The outputs of each body part recognizer could be fed into a top-level convolutional filter which checks for the existence of each body part near its expected position and generates output pixels which are likely to contain humans. Additionally, the face recognizer could be broken down into smaller filters which recognize small features such as eyes or the corners of a mouth before being convolved together to check for the existence of a face. Recognizing small objects from the raw input image and chaining multiple convolutions together in a hierarchy is much more robust because small features tend to be more rigid and are less likely to contain dramatic lighting differences.

A neural network is nothing more than a hierarchy of convolutions and other simple operations such as max pooling, thresholding, or sigmoid functions. Figure 2 shows a typical dataflow graph for a neural network. Each *layer* in the hierarchy recognizes features from its upstream neighbor by convolving its predecessors with a kernel for the feature it wants to recognize. The raw input to the neural network is typically referred to as the input layer with hidden layers and an output layer designating intermediate results and the final result respectively. Pooling essentially reduces the resolution of an image so that the detection of large features does not require large, computationally intensive, convolutional kernels. Pooling also gives a network tolerance for displacements between smaller features which make up larger features. For example, eyes may be farther apart on some faces but pooling would allow pixels

from multiple nearby locations to trigger the same actions downstream in the network. Since the purpose of a neural net is often to detect if an image contains some object, it makes sense that the entire image would need to be reduced gradually down to a single scalar value which measures the certainty that the image contains that object.

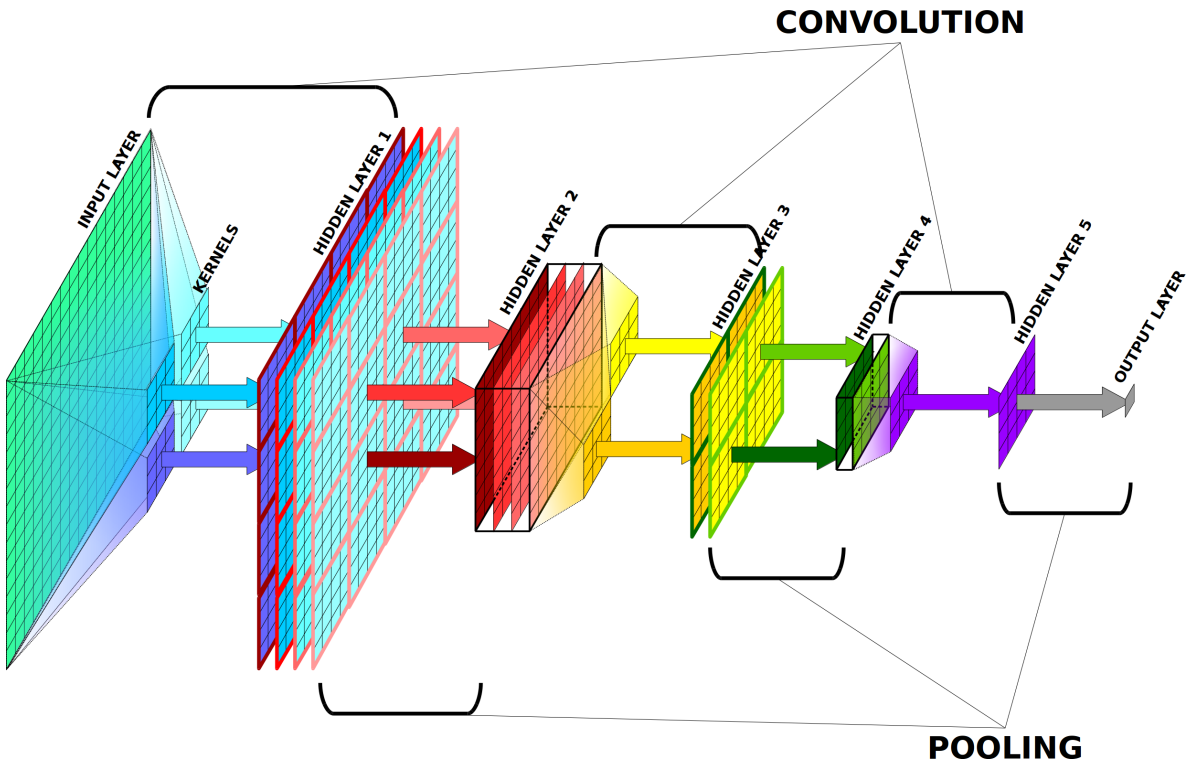


Figure 2. High level overview of a neural network built from convolutions and pooling. Good results can be obtained with only these two operations, but more advanced operations such as per-pixel sigmoids can enhance network accuracy further.

1.3 Training

Section 1.2 describes how a convolution kernel can help recognize certain features from an input image. However, for a kernel to effectively recognize a certain class of object, such as an eye, the kernel must somehow represent as many different variations of eyes as possible. A quick solution may be to create a database of many eyes and average the images together to produce a kernel. Unfortunately, this requires the programmer to micro-manage each stage of

the neural network by explicitly specifying what class of object each layer must detect. Additionally, building a convolution kernel from many images in this way does not harden the kernel against objects that are not eyes but look similar to eyes, called false positives.

A more flexible and automated method of generating convolution kernels is to *train* the kernel, as well as other neural network parameters, to classify a set of inputs for which the correct output is already known. To train a network to recognize an eye, the programmer would collect a large database of thousands of images where some, but not all, of the images contain an eye as the main feature. Second, the programmer would manually inspect each image and *label* it with a positive tag if it contains at an eye and a negative tag otherwise. Third, the programmer would develop some metric to measure the accuracy of the network. For the eye recognizer, a good accuracy metric would be the percentage of input images for which the neural network output matched the label for that image.

When the eye recognizer is trained, the convolution kernels and other parameters are initialized, usually with random values. After initialization, each input image is placed on the input layer of the neural network to generate a result at the output layer. The output result is compared to the correct label for that image and the accuracy metric is calculated. The initial accuracy will usually be very low. The training program will then perturb one or more neural network parameters and the accuracy metric will be re-evaluated. If the modified parameters produce a higher accuracy, the new parameters are used for further iterations. Otherwise, the parameters are reverted. This iterative training process continues for thousands or millions of iterations until an acceptable accuracy is attained.

This explanation of training is oversimplified in many ways. First, each training iteration typically does not consume the entire set of input images at one time, as this would be too computationally intensive. Instead, a small subset of the training data called a *batch* is selected and the optimal parameters for the batch are reached quickly. Multiple batches are run and their results are merged to produce the final neural network parameters. The full training flow is illustrated in figure 3.

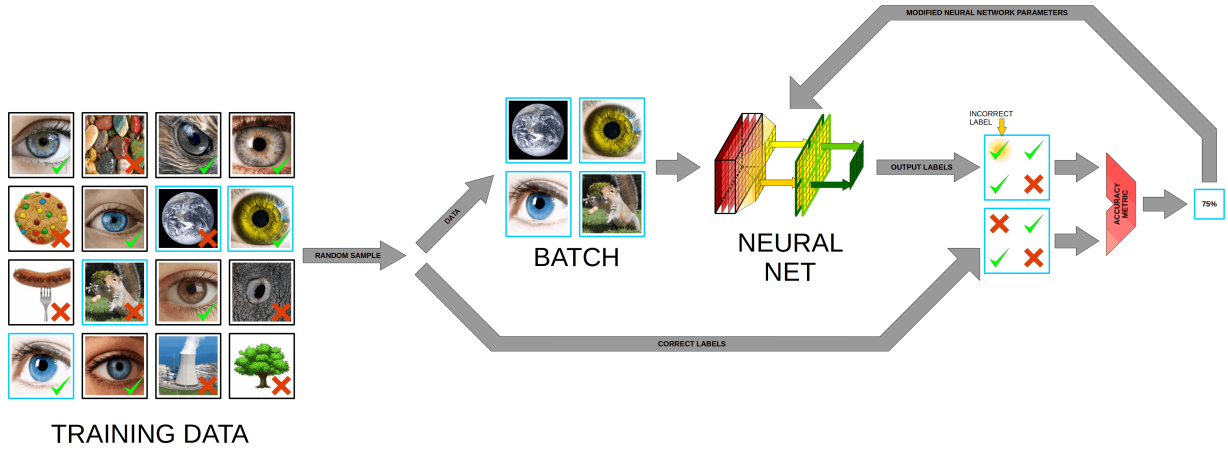


Figure 3. Typical raining setup for a neural network.

Furthermore, it is customary to split the training data and labels into two mutually exclusive sets called the training set and *validation* set. First, training is performed in the usual way using only the training data. Second, the accuracy metric is run using the validation set. In this way, the accuracy can be reported using data that the network had never seen before.

As a final aside, the values of each convolution kernel as well as any other trainable constants within the network are typically referred to as *weights*. So a 3x3 convolution kernel such as the one shown in figure 4 would constitute nine weights. Additionally, the data in each neural layer is typically restricted or normalized to floating point values between 0 and 1 so that each data item can be treated as a fuzzy Boolean value that is closer to true (1) if that pixel is centered on a matching object. In line with this fuzzy Boolean concept, each neuron typically has a single “bias” which defines the threshold above which an output is considered “true”.

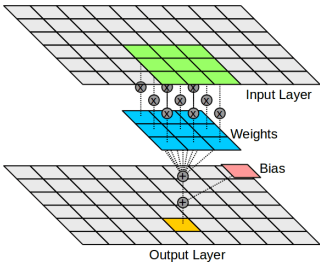


Figure 4. View of a single *neuron* within a neural network. Trained kernel values are typically called *weights* in neural network literature.

CHAPTER 2: BASIC FUNCTION APPROXIMATION

2.1 Overview

Chapter 1 builds the analogy of neural networks as trainable pattern recognition devices. Each convolution kernel represents the signature of some feature to be recognized and each layer in the network recognizes increasingly more complex features using the features recognized in the next upstream layer. This chapter will present a new analogy: that neural networks can be trained to approximate mathematical functions.

2.2 Piecewise Linear Approximation Using a Neural Network

Suppose you have a set of 2D coordinates on the X-Y plane and want to fit a curve to this data. Recall that the equation of a line can be written as:

$$y = mx + b \qquad \begin{array}{l} \text{m = The slope of the line} \\ \text{b = The y-intercept of the line} \end{array} \tag{1}$$

Recall that a typical neuron performs a convolution followed by an add. If we restrict the convolution kernel size to 1x1, then the neuron will perform a single multiply on the corresponding item from the input layer followed by the addition of a single bias. This is identical to the equation of a line written above, a multiply on the input followed by an add! This result may seem like a trivial result, since fitting a line to a set of points can be done more efficiently via other methods which existed before neural networks. The important result here is that the architecture of a neural net shown in figure 5 (really, a single neuron) has this capability without any special modifications to the neuron or training procedure. A tool originally designed for image recognition has been found to perform a new role in a very natural way.

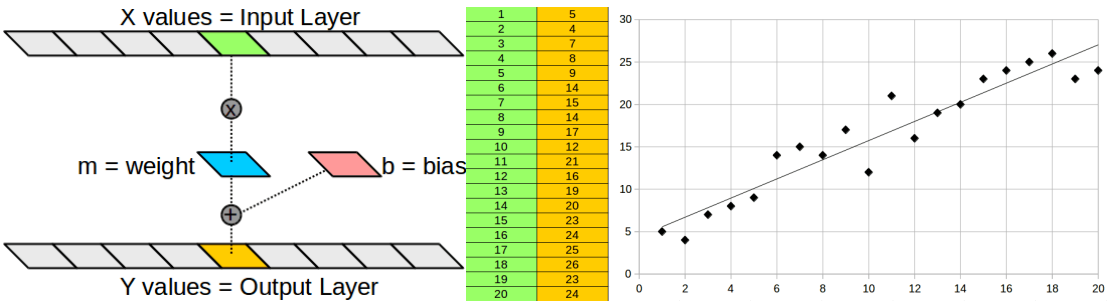


Figure 5. Using a neuron with a 1x1 convolution kernel to find a trend line.

Finding the equation of a single line is useful, but a neural network's true curve fitting power can be demonstrated when it is required to approximate an arbitrary function. It is possible to augment the previous correspondence between a convolutional neural network and the equation of a line to support piecewise linear functions. To illustrate how to do this, consider the following example in which the dataset of figure 6 can be approximated by the piecewise linear function of figure 7.

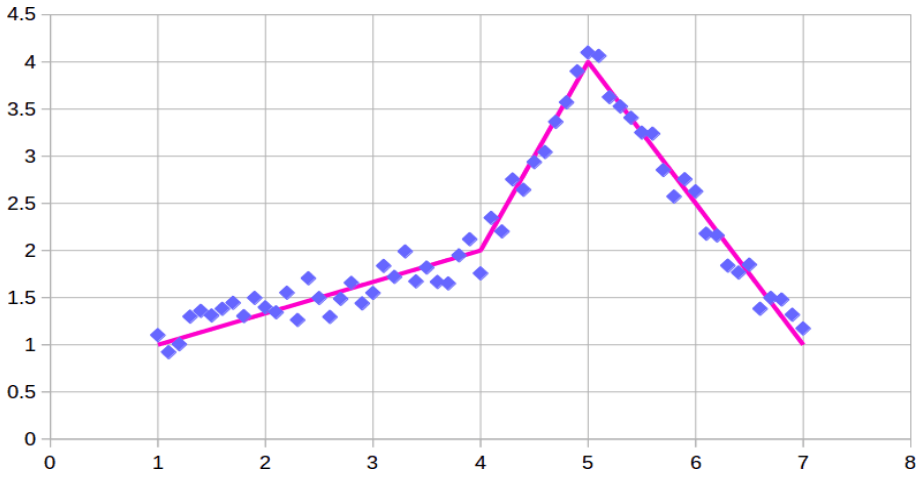


Figure 6. Scatter plot and piecewise linear function fitted to the data. Note that the neural network is only given the X-Y data in the scatter plot. The trend line shows the final result that we want to neural network to converge to on its own.

$$x=[4\text{ to }5] \qquad x=[1\text{ to }4] \qquad x=[5\text{ to }7]$$
$$y=\frac{1}{3}x+\frac{2}{3} \qquad y=2x-6 \qquad y=-\frac{3}{2}x+\frac{11}{3}$$

Figure 7. We want the neural network to converge to a piecewise linear function similar to this one.

$$y=\frac{1}{3}max(1,x)+\frac{5}{3}max(4,x)-\frac{7}{2}max(5,x)+\frac{23}{2}$$

Figure 8. Piecewise linear equation of figure 7 written using only adds, multiplies, and max operations.

We already have demonstrated that a neural network can be fitted to a single linear function, but have not yet explained how it is capable of selecting one of many functions within a certain region on the X-axis. Although a neural network can be built of arbitrary functions at each layer, it is more useful to demonstrate that the networks can perform a piecewise linear approximation using only "standard" operations which are commonly available as built-ins for neural network libraries and that are well-known to neural network programmers who often end up guessing which operations are at each layer in the network. One standard operation which was already been introduced in chapter 1 is $max(x,y)$, which is a special case of the more general max pooling operation. Figure 9 shows a dataflow diagram of a neural network that implements the equation of figure 8 using a max pooling layer followed by a convolution layer. Other implementations will be possible, but showing that at least one is possible using a reasonable architecture is sufficient to prove that neural networks are capable of approximating piecewise linear functions.

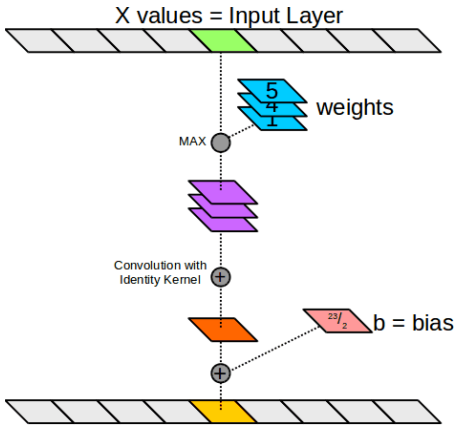


Figure 9. Post-training neural network implementation of the equation of figure 8.

Almost any continuous function can be approximated by a piecewise linear function. Since we have shown that a neural network can be trained to implement a piecewise linear function, then it follows that a neural network can be trained to approximate more arbitrary curves such as polynomials, sine waves, and ellipses.

2.3 Activation Functions

It turns out that performing the function $max(x,c)$, where c is a constant, is so useful that the operation has been given a special name: the *rectifier function*, or *relu*. Usually relu is defined to be:

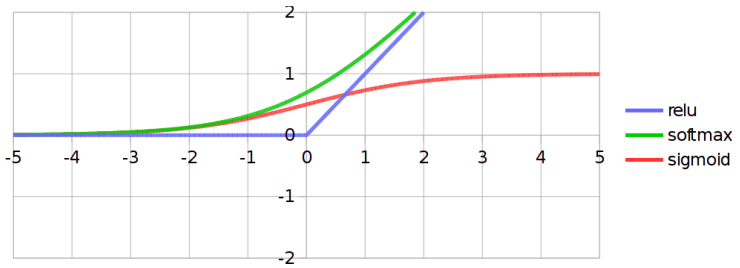
$$max(x,0)=relu(x) \tag{2}$$

It is simple to implement $max(x,c)$ using relu by using:

$$max(x,c)=relu(x-c)+c \tag{3}$$

The rectifier function is a member of a set of *activation functions*, shown in figure 10, which typically perform some sort of clamping operation. Other activation functions include

softmax and sigmoid functions. Sigmoid functions are especially useful for mapping some input to a fuzzy Boolean value between 0 and 1 such that the sigmoid function saturates to a value close to 0 or 1 rather quickly.



$$relu(x) = max(x, 0)$$

$$softmax(x) = \ln(1 + e^{x})$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Figure 10. Common activation functions.

The performance-minded reader may be concerned with the softmax and sigmoid functions, since they now require floating point division and exponentiation while relu can operate on either integer or floating point values using only a comparison and a branching instruction. This is a valid concern, and often neural network designers will use expensive functions excessively to perfectly match a mathematical model when a cheaper activation function would have been sufficient. However, there are many cases where using an expensive function such as sigmoid truly produces accuracy superior to that of a cheaper function. In these cases, it is possible to approximate a sigmoid using a lookup table and linear interpolation, since the general shape of the function is usually more important than the somewhat arbitrary mathematical formula used to generate it.

CHAPTER 3: CODE APPROXIMATION

3.1 Overview

After reading the material in chapter 1, the reader should now have a good intuition of how a neural network is structured and what its capabilities are. Chapter 2 builds a mathematical analogy for neural networks to compliment the feature recognition analogy presented in chapter 1.

Following from function approximation, it seems natural to try and approximate more arbitrary functions written in a software language such as C++. This is indeed possible and has been demonstrated first by Esmaeilzadeh [2] as well as in this document.

3.2 Code Approximation Infrastructure

Before a neural network can be used, the programmer must define its data-flow architecture. This means specifying the number of layers in the network, what operations are performed at each layer, kernel dimensions, and pooling window sizes. Once set, these dimensions are fixed during training and all subsequent operations on the neural network. Since it is not always clear what dataflow architecture is best suited for a particular task, it is common to define multiple neural network architectures, train each independently, and evaluate the accuracy and performance of each network after training is complete.

Once one or more neural network topologies are defined, we must train each network such that the weights and biases converge to a set of values which cause the input-output behavior of the neural network to match the input-output behavior of the original function as closely as possible. Before training can begin, the training dataset must be gathered. Recall that training data essentially consists of an “answer key,” or a set of function inputs and the corresponding correct outputs. Generating the training dataset can be easily accomplished by simply calling the function to be approximated on a wide variety of operands and dumping the

arguments and return values to a file. Traditional code development workflows already encourage test benches to be written for each function in a project, so creating the training dataset for a function within a well-managed project can be as easy as temporarily adding print statements before and after a function call within a test bench as shown in figure 11. However, even if a test bench has 100% coverage of a function, it may do so using the minimal number of test cases required to achieve this coverage. In these cases, the programmer should generate a large number of additional random function inputs and outputs for the training data.

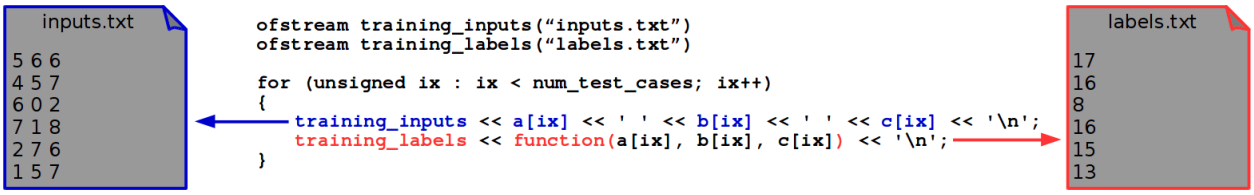


Figure 11. Generating training data for a C++ function is as simple as dumping its input arguments and return values to files.

Once the training data is generated, training can proceed in the usual way. The neural network input layer(s) should contain the same number of elements as function arguments. A clever neural network architect could also try to match the dimensions the input layer to match the dimensions of the arguments. For example, if a 2D texture is passed as an argument, the neural input layer should reserve two dimensions dedicated to that argument. Matching argument dimensions is not required in general but can make training converge more rapidly or with less data.

If a neural net can be trained to approximate a function with an acceptable level of accuracy, then the network can be run on a GPU, FPGA, or ASIC. The advantage here is that the programmer only needed to throw a few print statements around a CPU function rather than write an entire GPU program. The neural network topologies can also be imported from a library with ease. The disadvantage is that the neural network implementation will be less efficient than a hand-written GPU code, as it will typically operate on floating point numbers and contain additional unnecessary neural connections.

Not all functions can be mapped to a neural network. In particular, side effects such as system calls or variable-length arrays are not simple to implement with a single neural network. Although some reasonable workarounds exist, the best candidates for approximation will be pure data transformations. Table 1 outlines the criteria that a function must meet if it is to be replaced by a neural network:

Table 1. Criteria for Neural Network Approximation

Criterion	Description	Workaround
Fixed-size operands	The inputs and outputs of the function to be approximated shall be of a fixed size since a neural network has a fixed topology once instantiated.	Arrays of arbitrary length can be tiled before being fed into a neural network.
No side effects	A neural network can only apply data transformations. Printing to the screen or spawning threads cannot exist within the function to be approximated.	The function to be approximated may produce some data output which triggers side effects.
Error Tolerance	A neural network will not produce an input-output behavior identical to the original function. Some rare function inputs may produce extremely deviant outputs.	None

CHAPTER 4: EXPERIMENTAL RESULTS

4.1 Overview

This chapter summarizes a set of program code which was successfully approximated by a neural network and the steps taken to reach the presented results.

4.2 Approximating an Image Gradient

This example was deliberately chosen to be fairly trivial, since it was the first one attempted during research. The image gradient operation consumes a greyscale 2D image as an input and yields a greyscale output image of the same dimensions. For color images, the gradient operation could be repeated for each color channel. Each pixel in the output image is "whiter" when there is a large positive change in intensity along the X or Y direction across the corresponding neighborhood from the input image, and "blacker" when the change is strong in the negative X or Y direction.

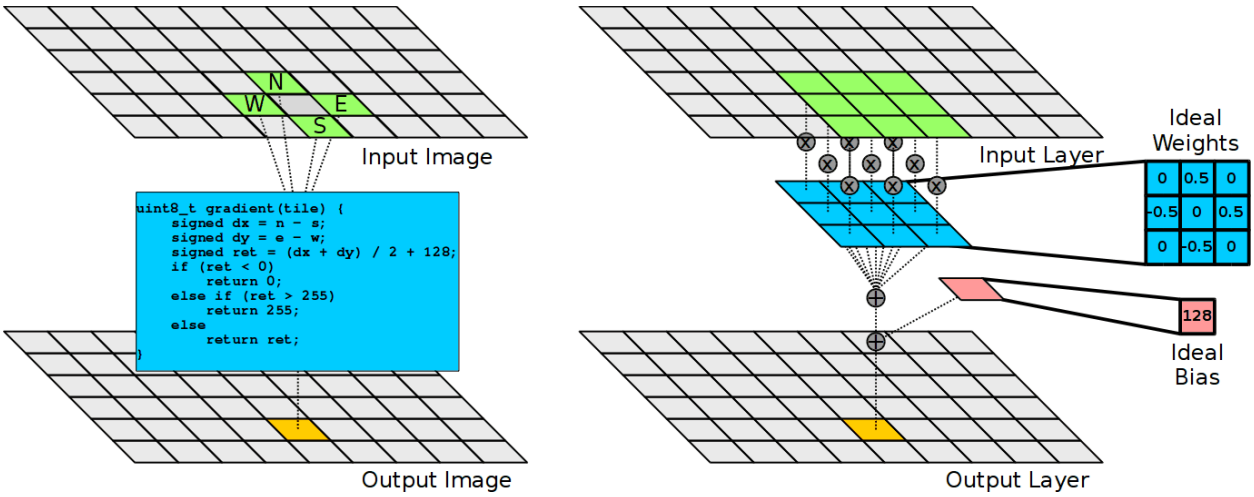


Figure 12. Image gradient operation and equivalent neural network.

The neural network chosen to approximate this gradient operation was chosen to be a single convolutional layer with 9 weights for a 3x3 convolution kernel and 1 bias per neuron. As an afterthought, it would have been ideal to also include one or two relu layers after the convolutional layer to catch the clamping behavior of the if/else chain at the end of the code in figure 12. Still, since overflows and underflows are rare, error rates as low 0.3328 as were still achieved on the validation dataset using the metric of equation 4. Note that an error rate of 1 would indicate that the average pixel in the neural output image had a value of 1 more or 1 less than the true value. Since each pixel can take a value from 0 to 255, an error rate of less than 1 is quite accurate.

$$error = |(\text{true\_output} - \text{neural\_output})| \tag{4}$$

This was the same metric used to guide the optimizer during training. Both the absolute difference and mean square error were used as accuracy metrics, but there was no significant difference in the required training dataset size, or final network accuracy. Therefore the easier to compute and more intuitive absolute difference metric was favored here.

After some trial and error, it was possible to make training converge to a convolution kernel and bias very close to the ideal one of figure 12. However, there were a few important practical details that needed to be acknowledged to get consistent training results:

1) The training data had to be generated using a random image such as the one shown in figure 13. Initial attempts traced the input-output behavior of the gradient function using a real-life image but yielded poor results. Generating an input image where each pixel was a random value between 0 and 255 yielded a more representative training set and better results.
2) The “training speed” or rate at which the optimizer was allowed to change each weight should to be very slow. Specifically, the optimizer was only allowed to change each weight by at most 0.01 during a single training iteration, which is on the order of 100x

less than the optimal values of the weights. Initial attempts with higher training rates did not properly converge but instead oscillated around sub-optimal weights.

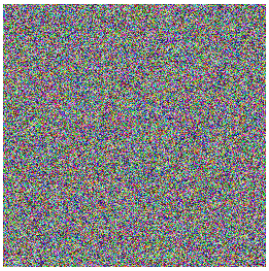


Figure 13. This white noise image yielded the best training data for the neural network because of its good coverage.

The first issue was fixed by using white noise images instead of realistic images to generate the training data. The underlying issue turned out to be that using a photograph to generate the training data did not yield enough variety of pixel colors nor gradient levels. Most natural looking images do not contain large gradients on average and therefore most of the resultant labels were around 128±16 even though the gradient function is capable of outputting values between 0 and 255.

Another, more subtle, problem also occurred when attempting to train the gradient approximation using a photograph instead of white noise to generate the training data. Training would often converge to a value such that validation would yield a high accuracy, but the operation actually performed by the neural network was completely incorrect. In the case of the gradient example, since most natural images do not contain many heavy gradients, the correct output gradient will be an almost uniform gray image. As a consequence, training would yield neural nets that would unconditionally output a uniform gray image where each pixel had the value 128. Technically, this achieved a “high accuracy” using the given metric, but this was obviously not the desired behavior. These bogus neural networks would also have random values for their weights and biases. Switching to a training set generated from a white noise image immediately rectified, yielding the results of figure 14. Note that the input image for figure 14 is property of CCP games [1].

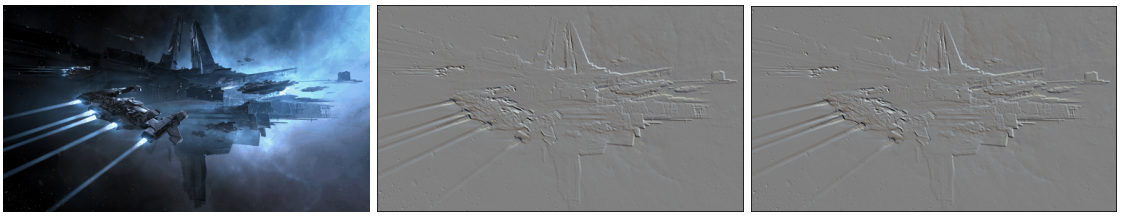


Figure 14. Visualization of the gradient operation on an input image (left). The true image gradient (middle) is visually indistinguishable from the neural network generated gradient (right). Input image source: CCP [1].

4.3 Approximating a Histogram

As a review, a histogramming function takes a 1D input vector of integers and generates a 1D output vector of integers, referred to as “bins”. Each element in the output vector (each bin) contains a count of all the input integers which fall within a certain range. For example, bin 0 would contain a count of how many input integers contained values between 0 and 15. Bin 1 would count the number of input integers between 16 and 31 and so on.

Approximating a function which computes a histogram is tricky, since it is not obvious how it could be implemented efficiently using any of the neural networks described thus far. Nevertheless, a simple 3 layer network consisting of a fully connected layer, followed by sigmoid function, then another fully connected layer yielded an accurate histogram approximation. Situations such as this where the programmer tries to implement a neural network without knowing all the internal details are quite common, especially for complex code blocks. It is still helpful for the programmer to be aware that the histogramming operation will require some type of sorting or thresholding intrinsic, and include a sigmoid layer within the neural network to provide this capability.

The error function used for training and validation is defined in the following way. For each input integer *n*, the neural network will guess which bin that integer belongs in. More specifically, the neural network will guess the index of the bin for a given integer. This bin index is then compared to the true bin index. For example, if the neural network assigns the input 78 to bin 5 but the correct bin was 4, the error for that single input integer will be 1. During training

and validation, the neural network will classify a large vector of inputs, and the average error metric will be computed for that batch.

$$error = mean(|neural\_bin\_ix(n) - true\_bin\_ix(n)|) \tag{5}$$

Care had to be taken when evaluating the accuracy of the neural network. Just like the image gradient experiment, it was common for training to yield a useless neural network which did not generate a histogram but still attained low values for the error metric. Typically, these bogus networks would output a uniformly distributed histogram, which was technically correct for the fairly large and uniform input dataset, but failed for small or nonuniform datasets.

As an additional sanity check, a histogram of the error metric was generated and reported in figure 15. Note that this histogram has nothing to do with the fact that this benchmark was also a histogram. The metric would report the number of times that the neural network's bin assignment matched the true bin, the number of times that the neural network was off by 1 bin index, and by 2 bin indexes, etc. This error distribution clearly showed that trained neural network did indeed assign the correct bin most of the time and that the error dropped off rapidly.

Error	Occurrences
0	41412
1	38961
2	34692
3	28602
4	22311
5	16347
6	11137
7	7001
8	4132
9	2453
10	1319
11	672
12	333
13	168
14	97
15	52
16	18
17	11
18	6
19	0
20	3
21	1

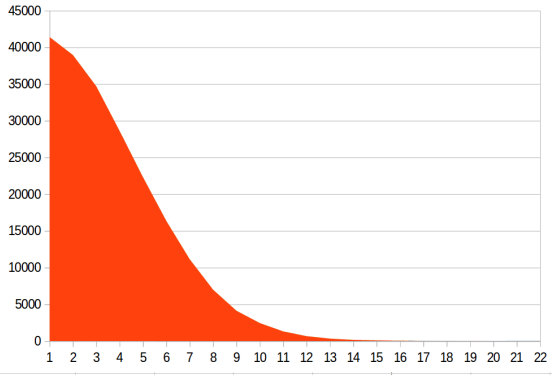


Figure 15. Distribution of all errors for validation set for neural net histogram approximation.

4.4 Approximating a Heat Flow Simulation

The most complex block of code that was approximated using a neural network was a 2D thermal simulation called “Hotspot” which comes packaged as part of the Rodinia benchmark suite. The simulation quantizes the 2D heat map into a grid where each point on the grid is essentially a pixel and each pixel contains its own temperature and power value. After initialization, the simulation is run for a desired number of time steps. Each timestep, the value of each pixel is computed using its temperature and power value from the previous step. Positive power values add heat to the system through the corresponding pixel and negative power values cool the corresponding pixel. The rate of heating or cooling is proportional to the magnitude of the power value. Additionally, the temperature values (but not power values) from neighboring pixels are consumed from the previous timestep to generate the next value. By examining a neighborhood around a pixel, extreme temperature gradients lead to faster heating or cooling. Additional parameters such as the thermal conductivity of the entire grid are also available for manipulation.

Training data for Hotspot was generated by dumping the input tile around each output pixel for both the temperature grid and power grid. The output pixel itself formed the corresponding label. The thermal conductivity coefficient was omitted from the training data for simplicity. The most accurate topology for this benchmark was found to be a single convolutional layer with a very small 3x3 kernel size. The trained network achieved very low average error rates below 1.0 using the same error metric as equation 4. Since this physics simulation has some similarities to the image gradient operation, superior training accuracies were achieved by generating a white-noise heat map to use as training data. Initially, a realistic CPU temperature map and power map were used, but they were too uniform across large areas of the chip and caused overfitting.

4.5 Optimizations and Future Work

Since the dataflow graphs associated with neural networks are typically very periodic and embarrassingly parallel, they are perfectly suited for execution on a GPU, even if the original code was not particularly uniform. Therefore, it is very beneficial to consolidate multiple layers from a neural network such that they read and write to fast local memories of a thread block (CUDA) or work group (OpenCL). The 2D temperature grid is divided into tiles such that an input tile and output tile fit into the fast scratchpad memory local to a thread block or work group. To honor dependencies, the output tile is always smaller than the input tile since convolutions at the edge of the tile have an input window which overlaps a neighboring window. The more convolutions are consolidated into a single tile, the larger this input halo needs to be and the fewer output pixels are computed each run. Therefore there is a trade-off in which more consolidation decreases costly accesses to slow global GPU memory but increases the amount of redundant computation done on halo pixels.

As a pilot experiment, the Hotspot benchmark was run with multiple iterations per global memory transaction. Since Hotspot was written in CUDA, this meant tiling the temperature grid into thread blocks and performing multiple iterations within shared memory. Note that the data reported in figure 16 is for the original Hotspot code; the neural network implementation was not yet ready to perform this optimization at the time this document was written. The results of this experiment show how consolidation reduces end-to-end execution time at first, but the number of redundant computations from halo pixels soon overtakes the memory bandwidth benefits.

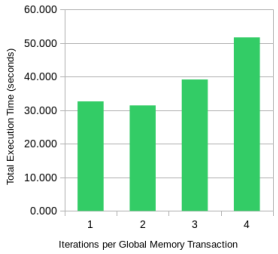


Figure 16. Performing more simulation iterations within CUDA shared memory increases memory bandwidth, but also increases the number of redundant computations.

As a future work, it may be possible to break large convolution kernels into a sequence of multiple small convolution kernels. This will decrease the accuracy of the neural network, since very few convolution kernels are perfectly separable, but it will reduce the number of computations for an equally sized dependency window.

As an additional future work, it may be possible to increase the performance of the Hotspot approximation by using a recurrent neural network (RNN) to keep the temperature state within the neuron and potentially avoid some shared memory transactions. A recurrent neural network is very similar to the networks presented thus far, but they also contain integrated memory within each neuron. A detailed discussion of RNNs is beyond the scope of this document, but the reader may wish to refer to Olah [3] for an in-depth RNN tutorial.

REFERENCES

[1] *Astrahus Citadel*. 2016. *EvE Online Updates*. Web. 11 Apr. 2017. <http://updates.eveonline.com/tag/structures/>.

[2] Esmaeilzadeh, Hadi, Adrian Sampson, Luis Ceze, and Doug Burger. "Neural Acceleration for General-Purpose Approximate Programs." 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (2012): n. pag. Web.

[3] Olah, Christopher. "Understanding LSTM Networks." *Understanding LSTM Networks -- Colah's Blog*. N.p., 27 Aug. 2015. Web. 11 Apr. 2017.

[4] *Tensorflow*. Program documentation. *MNIST For ML Beginners*. Vers. 0.12. Google, 8 Mar. 2017. Web. 11 Apr. 2017. <https://www.tensorflow.org/get_started/mnist/beginners>.

APPENDIX A: GLOSSARY

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
Batch	A subset of the input dataset used to train a neural network
CPU	**C**entral **P**rocessing **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DSP	**D**igital **S**ignal **P**rocessor
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
Kernel	A small image that is repeatedly overlapped with tiles from a larger image during a convolution operation
Label	Some metadata associated with the input data for a neural network describing the desired or correct output of the neural network
Neural Network	A dataflow graph which recognizes features features from an input and for which the parameters of each operation within the dataflow graph are trainable
Rectifier Function	An activation function of a real number *x* such that $relu(x) = max(x, 0)$
Sigmoid	An activation function of a real number *x* such that $sigmoid(x) = \frac{1}{1+e^{-x}}$
Softmax	An activation function of a real number *x* such that $softmax(x) = \ln(1+e^{x})$
RELU	Abbreviation for “Rectifier Function”
Training	The process of fitting neural network parameters to a set of input objects for which the correct output objects is already known
Validation	The process of measuring the accuracy of a neural net using data that was not used during training
VLIW	**V**ery **L**ong **I**nstruction **W**ord

APPENDIX B: SOFTWARE LIST

[1] *CUDA*. Computer software. *CUDA Downloads*. Vers. 8.0. NVIDIA Corporation, 28 Sept. 2016. Web. 11 Apr. 2017. <https://developer.nvidia.com/cuda-downloads>.

[2] *GPGPU Sim*. Computer software. University of British Columbia, n.d. Web. 11 Apr. 2017. <http://www.gpgpu-sim.org/>.

[3] *Rodinia*. Computer software. *Rodinia*. Vers. 3.1. University of Virginia, 12 Dec. 2015. Web. 11 Apr. 2017. <http://lava.cs.virginia.edu/Rodinia/>.

[4] *Tensorflow*. Computer software. *Tensorflow*. Vers. 0.12. Google, 9 Nov. 2015. Web. 11 Apr. 2017. <https://www.tensorflow.org/>.